CSE166_FA23_assignment_3

October 23, 2023

1 CSE 166 Image Processing, Fall 2023 - Assignment 3

Instructor: Ben Ochoa

Assignment due: Mon, Oct 30, 11:59 PM

Name:

PID:

1.1 Instructions

1. All solutions to the problems below must be written in this notebook. Programming aspects of the assignment must be completed using Python (preferably 3.10).
2. This assignment must be completed **individually**. For more details, please review the Academic Integrity Policy and Collaboration Policy on Canvas.
3. The packages you need to complete the assignment will be indicated in the problem descriptions.You are not allowed to use the packages that directly solve the problems. Feel free to ask the instructor and TAs if you are unsure about the packages to use.
4. It is highly recommended that you begin working on this assignment early.
5. After finishing the assignment in the notebook, please export the notebook as a PDF and a python source file. You need to **submit 3 files: the Notebook, the PDF and the python file** (i.e. the `.ipynb`, the `.pdf` and the `.py` files) on Gradescope.

- To convert the notebook to PDF, you can choose one way below:
  - You may first export the notebook as HTML, and then print the web page as PDF
    - e.g., in Chrome: File → Save and Export Notebook as → "HTML"; or in VS-code: Open the Command Palette by pressing Ctrl+Shift+P (Windows/Linux) or Cmd+Shift+P (macOS), search for Jupyter: Export to HTML
    - Open the saved web page and right click → Print... → Choose "Destination: Save as PDF" and click "Save")
  - If you have XeTex installed on your machine, you may directly export the notebook as PDF: e.g., in Chrome, File → Save and Export Notebook as → "PDF"
  - You may use nbconvert to convert the ipynb file to pdf using the following command `jupyter nbconvert --allow-chromium-download --to webpdf filename.ipynb`

- To convert the notebook to python file, you can choose one way below:
  - You may directly export the notebook as py: e.g., in Chrome, File → Save and Export Notebook as → "Executable script"; or in VScode: Open the Command Palette and search for Jupyter: Export to Python Script
  - You may use nbconvert to convert the ipynb file to python file using the following command `jupyter nbconvert --to script filename.ipynb`

6. Please make sure the content in each cell (e.g. code, output images, printed results, etc.) are clearly visible and are not cut-out or partially cropped in your final PDF file.
7. While submitting on gradescope, please make sure to assign the relevant pages in your PDF submission for each problem.

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

2 Problem 1: Textbook problems (11 points)

Download the following textbook problems from the Resources tab on Piazza. **Answers to different versions of these problems will not be accepted.**

These textbook problems are conceptual and/or math problems, not programming problems. You are provided with a Markdown cell to answer each problem. Do not change this cell to a Code cell or create a new Code cell. **Answers in the form of code will not be accepted.**

2.1 a) Problem 4.3 (1 point)

Your answer here:

2.2 b) Problem 4.4 (1 point)

Your answer here:

2.3 c) Problem 4.9 (2 points)

Your answer here:

2.4 d) Problem 4.27 (1 point)

Your answer here:

2.5 e) Problem 4.33 (2 points)

Your answer here:

2.6 f) Problem 4.40 (3 points)

Your answer here:

2.7 g) Problem 4.45 (1 point)

Your answer here:

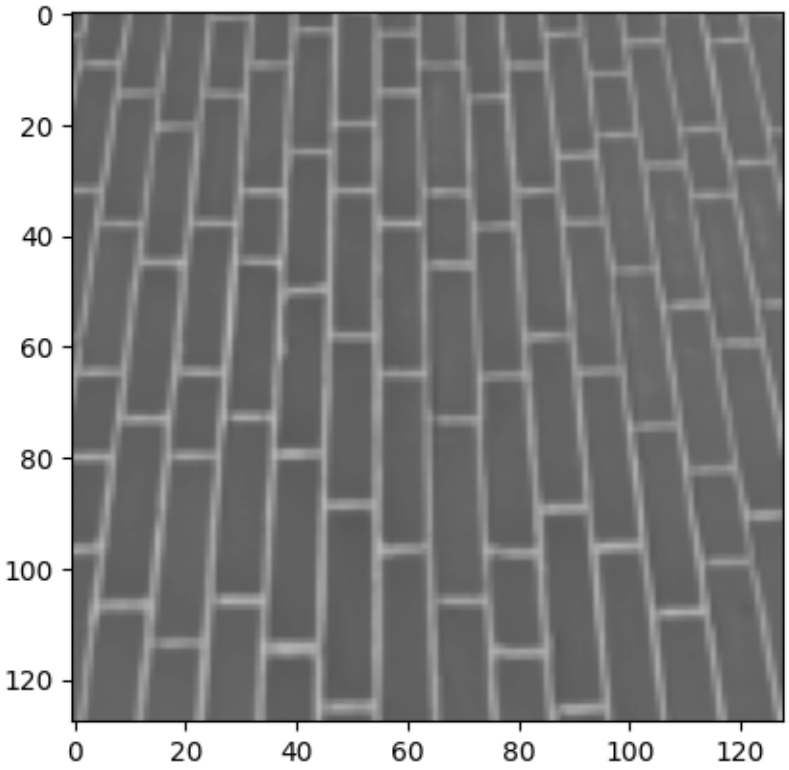
3 Problem 2: Programming: The Fourier Transform and Filtering in the Frequency Domain (35 points)

3.1 Part 1: The Fourier Transform Pair (10 points)

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data
from skimage.transform import resize
from skimage.color import rgb2gray
import cv2
import time
```

[ ]:
```
# Read and display image
img = 255 * resize(data.brick(),(128,128)) # resize the image to a smaller size␣
 ↪to reduce the computational time

plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show()
```



1) Complete the function **dft2d** that computes the 2D discrete Fourier transform of an array. The function must take as input a 2D array and return as output the 2D array of complex numbers corresponding to the discrete Fourier transform.

[ ]:
```
# function that computes the 2D discrete Fourier transform of an array.
def dft2d(arr):
  """
  arr: input array (H,W)

  returns:
  F: 2D discrete Fourier transform of the input array (H,W)
  """
  # your code here
  F = []

  return F
```

2) Generate a random numpy array of size 100×100 using `numpy.random.rand`. Apply Fourier transform to the random array by using your **dft2d** function and numpy.fft.fft2 function. Print the execution time of the two functions.

[ ]:
```
"""
Call the function dft2d and np.fft.fft2 with a random 100*100 array
Compare the execution time of your dft2d function with the numpy.fft.fft2␣
 ↪function
"""
# your code here
```

3) Why do you think the execution times are different?

Your answer here:

4) Complete the function **dft_ift** which, in order,
   - Calls the **dft2d** function with the image to compute the discrete Fourier transform (DFT) $F(u, v)$ (shifted such that the zero-frequency component $F(0, 0)$ is centered) of the input image $f(x, y)$
   - Calculates the magnitude $\|F(u, v)\|$ and phase $\phi(u, v)$ of $F(u, v)$
   - Calculates the DFT $G(u, v) = \|F(u, v)\|e^{j\phi(u,v)}$
   - Computes the inverse discrete Fourier transform (IDFT) $g(x, y)$ of $G(u, v)$ (after inverting the centering shift)
   - Returns the real part of g(x,y), magnitude $\|F(u, v)\|$ and phase $\phi(u, v)$ of $F(u, v)$

Note: You may use the NumPy functions np.fft.fftshift, np.fft.ifftshift, np.fft.ifft2

[ ]:
```
# see part 1 above to read complete function description
def dft_ift(img):
  """
  img: input image (H,W)
```

```
    returns:
    img_g_real: real part of g(x,y) (H,W)
    f_mag: magnitude ||F(u,v)|| (H,W)
    f_phase: phase of F(u,v) (H,W)
    """
    # your code here
    img_g_real = []
    f_mag = []
    f_phase = []

    return img_g_real, f_mag, f_phase
```

5) Call the **dft_ift** function with the brick image. Display the input and the output image and set the value range to [0, 255]. Display figures of the log magnitude $\log\|F(u,v)\|$ and phase $\phi(u,v)$ of the discrete fourier transform $F(u,v)$, leave the value range as default ([min, max]).

[ ]:
```
"""
Call the dft_ift function with the brick image. Display the input and the␣
 ↪output image.
Additionally, display figures of ||F(u,v)|| and phase(u,v).
"""
# your code here
```

6) What are the row and column indices of the $F(0,0)$ component before and after the centering shift?

Your answer here:

3.2 Part 2: The Convolution Theorem (15 points)

The objective of this problem is to show that filtering of an image by convolving it with a kernel can be done in both spatial domain and frequency domain.

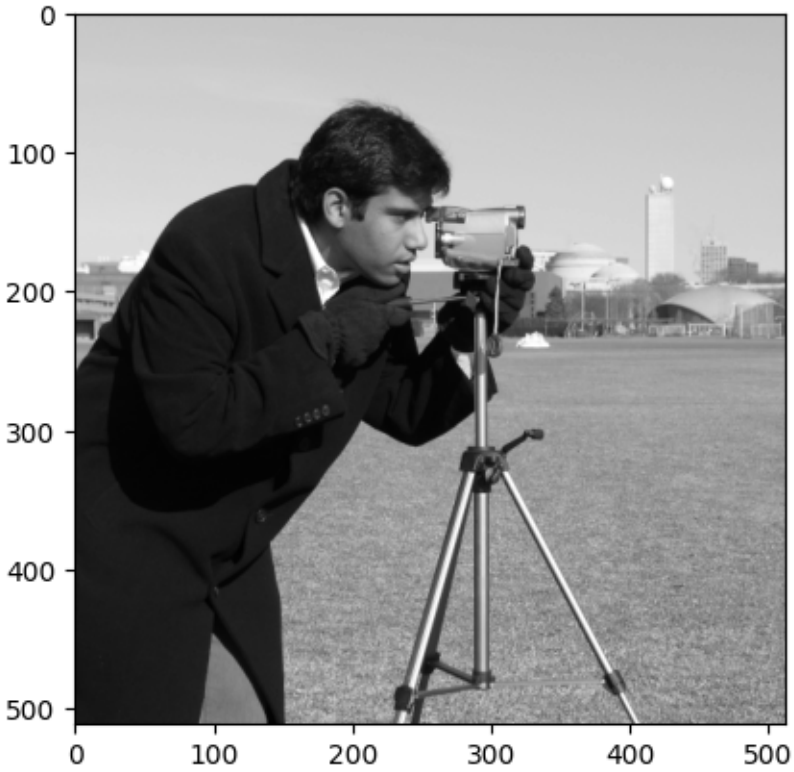
The output image

$$g(x,y) = f(x,y)\ h(x,y) = \mathfrak{F}^{-1}\{F(u,v)H(u,v)\}$$

where $F(u,v)$ and $H(u,v)$ are the DFTs of the input image $f(x,y)$ and kernel $h(x,y)$, respectively.

[ ]:
```
# Read and display the image
img = data.camera()

plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show()
```



1) Complete the function **filter_in_frequency_domain** that applies a filter to an image in the frequency domain. The function inputs are a grayscale image and a kernel, and the function output is the filtered (floating point) image corresponding to the input image. The inputs and output are in the spatial domain. Zero padding must be used to mitigate wraparound error. The calculated output image must be the same size as the input image.

Notes:

- You may use the NumPy functions np.fft.fft2, np.fft.ifft2, np.fft.fftshift, np.fft.ifftshift
- Implement padding by yourself, do not use np.pad.
- Ideally, after the inverse Fourier transform, the imaginary part of the result will be 0. However, due to floating-point error, the imaginary part will consist of values that are nearly 0. As such, only use the **real part** of the of the filtered image as the output.

[ ]:
```
# function that applies a filter to an image in the frequency domain
def filter_in_frequency_domain(img, kernel):
  """
  img: input image (H, W)
  kernel: filter (kH, kW)

  returns:
  f_img: image filtered in frequency domain (H, W)
  """
```

```
  # your code here
  f_img = []

  return f_img
```

2) Complete the function **conv_theorem** that applies the filter to the image $f(x, y)$ in the frequency and spatial domain. The function takes as input an image and a kernel, and returns the output images filtered in the frequency and spatial domain.

Notes: - You must call the function **filter_in_frequency_domain** to apply the filter in the frequency domain. - You may use the function cv2.filter2D to apply the filter in the spatial domain.

[ ]:
```
# function that applies the filter to the image in the frequency and spatial␣
 ↪domain.
def conv_theorem(img, kernel):
  """
  img: input image (H,W)
  kernel: kernel (kH, kW)

  returns:
  f_img: image filtered in frequency domain (H, W)
  sp_img: image filtered in spatial domain (H, W)
  """
  # your code here
  f_img = []
  sp_img = []

  return f_img, sp_img
```

3) Call the function **conv_theorem** with the Laplacian kernel $h(x, y)$ and the camera image $f(x, y)$, where

$$h(x, y) = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Display the input image along with both resulting filtered images. Set the value range to [0, 255] for the input image, [-255, 255] for the output images. Calculate the root mean square error between the output images filtered in the frequency and spatial domain. Print the result.

[ ]:
```
"""
Call the function conv_theorem with the Laplacian kernel and the camera image. ␣
 ↪
Display the input image along with both resulting filtered images.
"""
# your code here
```

4) Briefly discuss your results. Why would it be beneficial to implement filtering in the frequency domain?

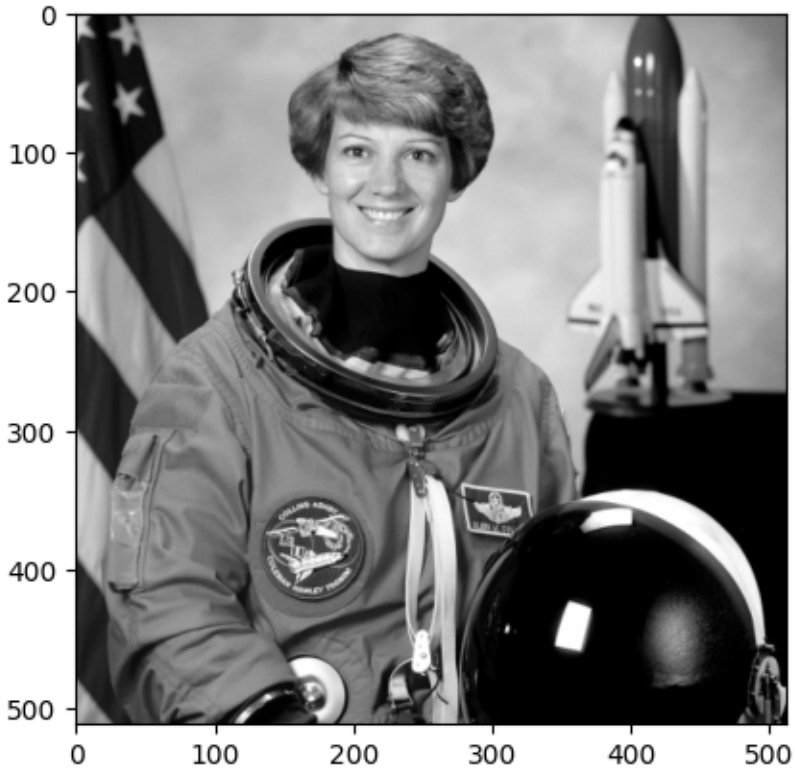
Your answer here:

5) (Optional): Subtract the edge image from the input image to yield a sharpened image. Display the result.

```
[ ]: # your code here (optional)
```

3.3 Part 3: Lowpass Filtering (10 points)

```
[ ]: # Read and display the image
img = 255 * rgb2gray(data.astronaut())
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.show()
```



1) Complete the function **ideal_lpf** that applies an ideal lowpass filter in the frequency domain. The function inputs are an image and a radius representing the cut-off frequency, and the function output is the image with ideal low-pass filter applied in the frequency domain.

Notes:

- You may use the NumPy functions np.fft.fft2, np.fft.ifft2, np.fft.fftshift, np.fft.ifftshift
- Ideally, after the inverse Fourier transform, the imaginary part of the result will be 0. However, due to floating-point error, the imaginary part will consist of values that are nearly 0. As such, only use the **real part** of the of the filtered image as the output.
- Do not convert the output image as uint8, leave it as float64.

```
[ ]: # function that applies ideal low-pass filter in the frequency domain
def ideal_lpf(img, radius):
  """
  img: input image (H,W)
  radius: Radius for the ideal lowpass filter

  returns:
  out: output image after low-pass filter has been applied in the frequency␣
 ↪domain
  """
  # your code here
  out = []

  return out
```

2) Complete the function **gaussian_lpf** that applies a Gaussian lowpass filter in the frequency domain. The function inputs are an image and a variance for the Gaussian filter, and the function output is the image with Gaussian low-pass filter applied in the frequency domain.

Notes:

- You may use the NumPy functions np.fft.fft2, np.fft.ifft2, np.fft.fftshift, np.fft.ifftshift
- Ideally, after the inverse Fourier transform, the imaginary part of the result will be 0. However, due to floating-point error, the imaginary part will consist of values that are nearly 0. As such, only use the **real part** of the of the filtered image as the output.
- Do not convert the output image as uint8, leave it as float64.

```
[ ]: # function that applies gaussian low-pass filter in the frequency domain
def gaussian_lpf(img, var):
  """
  img: input image (H,W)
  var: variance for the gaussian lowpass filter

  returns:
  out: output image after low-pass filter has been applied in the frequency␣
 ↪domain
  """
  # your code here
  out = []

  return out
```

3) Call the function **ideal_lpf** with the astronaut image for three different radii $D_0 = 25, 50, 75$. Display the original image and all three ideal lowpass filtered images. Set the value range to [0, 255] on all display calls.

```
[ ]: """
Call the function `ideal_lpf` with the astronaut image and for three different
radii = 25, 50, 75. Display the original image.
```

```
"""
# your code here
```

4) Briefly discuss your results.

Your answer here:

5) Call the function **gaussian_lpf** with the astronaut image and for three different variances $\sigma^2 = 21^2, 42^2, 64^2$. Display the original image and all three Gaussian lowpass filtered images. Set the value range to [0, 255] on all display calls.

[ ]:
```
"""
Call the function **gaussian_lpf** with the astronaut image and for three␣
↪different variance.
Display the original image and all three Gaussian lowpass filtered images.
"""
# your code here
```

6) Briefly discuss the difference between your Gaussian lowpass filtering results and your ideal lowpass filtering results.

Your answer here: